SANDIA REPORT

SAND2002-0675
Unlimited Release
Printed April 2002

Umbra's High Level Architecture (HLA) Interface

Eric Gottlieb, Michael McDonald, Fred Oppel

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

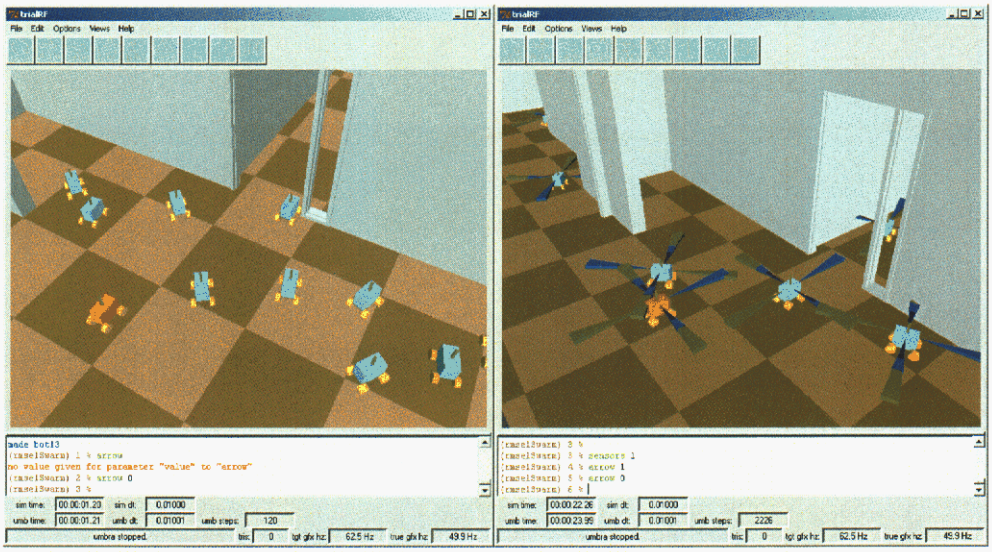
Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/ordering.htm

SAND2002-0675
Unlimited Release
Printed April 2002

Umbra's High Level Architecture (HLA) Interface

Eric Gottlieb, Michael McDonald and Fred Oppel
Intelligent Systems and Robotics Center
Sandia National Laboratories
Albuquerque, NM 87185-1004



Abstract

This report describes Umbra's High Level Architecture HLA library. This library serves as an interface to the Defense Simulation and Modeling Office's (DMSO) Run Time Infrastructure Next Generation Version 1.3 (RTI NG1.3) software library and enables Umbra-based models to be federated into HLA environments. The Umbra library was built to enable the modeling of robots for military and security system concept evaluation. A first application provides component technologies that ideally fit the US Army JPSD's Joint Virtual Battlespace (JVB) simulation framework for Objective Force concept analysis. In addition to describing the Umbra HLA library, the report describes general issues of integrating Umbra with RTI code and outlines ways of building models to support particular HLA simulation frameworks like the JVB.

INTENTIONALLY LEFT BLANK

Contents

Figures

Introduction

Umbra, Sandia's new modeling and simulation framework, links together heterogeneous collections of modeling tools to allow tradeoff analyses of complex robotic systems concepts. The Umbra framework allows users to quickly build models and simulations for intelligent system development, analysis, experimentation, and control. The models in Umbra include 3D geometry and physics models of robots, devices and their environments. Model components can be built with varying levels of fidelity and readily switched to allow models built with low fidelity for conceptual analysis to be gradually converted to high fidelity models for later phase detailed analysis.

This report describes Umbra's High Level Architecture (HLA) library. This library serves as an interface to DMSO's RTI NG1.3 software library and enables Umbra-based models to be federated into HLA environments. The library was built to enable the modeling of robots for military and security system concept evaluation. A first application provides component technologies that ideally fit the US Army JPSD's Joint Virtual Battlespace (JVB) simulation framework for Objective Force concept analysis. In addition to describing the Umbra HLA library, the report describes general issues of integrating Umbra with RTI code and outlines ways of building models to support particular HLA simulation frameworks like the JVB.

HLA Overview

As described in [Kuhl-99]

> *The HLA is a software architecture for creating computer models or simulations out of component models or simulations. The HLA has been adopted by the United States Department of Defense (DoD) for use by all its modeling and simulation activities. The HLA is also increasingly finding civilian application.*

The HLA is defined by three components: (1) Federation Rules, (2) the HLA Interface Specification, and (3) the Object Model Template (OMT). The DMSO has developed and supports a software library called the Run Time Infrastructure, or RTI, which implements the HLA interface specification and facilitates building HLA compliant codes. The RTI interface supports all inter-process communications in an HLA federation. Codes, like Umbra, interact with the RTI, and the RTI, in turn, exchanges data between federates.

Umbra Overview

Umbra simulations typically model devices and the environments within which they operate. These devices are modeled in Umbra as embodied agents, and fine and coarse-grained physical effects models are combined to represent interactions among devices and the physical world. Three-dimensional graphics displays are

used for visualization. Umbra can also be used to model disembodied agent systems to support basic research in distributed intelligence.

A key attribute of Umbra is its ability to correctly model the topological structure of integrated systems. For example, robots are typically modeled with behavior, effectors and sensors each represented within separate computational modules. (Here, effector models typically include vehicle motion as well as radio transmissions and other effects modules. Sensor modules typically include geometric sensors such as touch and proximity sensors as well as radio receivers and chemical sensor models.) These effector, sensor, and behavior modules are then configured into meta-modules that are connected in the same way that real sensors and effectors are connected to robot controllers.

Umbra fills a unique niche in modeling and simulation by addressing the "middle layer" of simulation fidelity. While typical Umbra simulations run at a mid-level of fidelity, Umbra simulations can and do range in fidelity from high level mission analysis tools (at a level similar to DoD constructive M&S tools like OneSAF and JCATS) to high fidelity engineering analysis tools (similar to MATLAB and ADAMS). In addition, a single Umbra simulation can incorporate models throughout this wide range of fidelities. In federations, Umbra can model intelligent systems and bridge low-level engineering with high-level constructive environments.

Generally, Umbra incorporates the following capabilities:

- Complex, non-linear world modeling – Umbra models geometry, physics, control laws, sensors, communication, functional subsystems and environments in a modular fashion. This enables the use of models with asymmetric levels of fidelity.
- System-level modeling – modules are configured to mimic system structures.
- Embodied agent modeling – Entities modeled with behavior, geometry, sensing, and physics. Robots are typically modeled as embodied agents.
- Disembodied agent modeling – Entities predominantly modeled with behavioral, as opposed to physical aspects. Typically used to analyze large collective systems of computational agents.
- Encapsulation – Enables modularity and legacy code integration.
- Continuous time and event driven simulation – Allows combining realistic simulation of real-world physics and control laws with high-level commanded event responses.
- Computational steering – Allows users to interactively modify simulations to highlight effects that develop during analysis. Adding unexpected obstacles to terrain models to examine dynamic control response is an example.
- Rapid integration of terrain and feature data – Allows analysis of systems in outdoor terrains and urban environments. Feature data

includes obstacle geometries, roads, and mobile vehicles as well as chemical plumes and other sensed physical features.

- Implementation uses C++ for compute-intensive tasks and Tcl/Tk[1] for application scripting and for its graphical user interface.

Modeling Robotic Behavior in Umbra

Robotic behavior is typically a result of complex interactions between the robot and its environment that result from the robot performing defined tasks. Umbra models are composed to match the target simulation environment framework and implementation. Within Umbra, robotic modules are built or composed as a collection of modules that separately model the robot's behavior and controls algorithms, the sensors that the robot's behaviors use, and the physics or motion of the vehicle as it interacts with its environment. The collection is often referred to as a meta-module. Additional communications modules are used to provide command interfaces and status outputs as well as to model how collections of robots communicate with one another.

For devices where behavior is independent of the physical environment, the physics and sensor modules are implemented as services that are detached from platforms or sensors. For example, intelligent information processing systems that do not move would not have sensors and physics.

Figure 1 shows how these meta-modules are organized within Umbra. It is noted that robots typically use several sensors to control their behavior. A shadowed box titled Robotic Sensors is used to represent this plurality. In addition, robot platforms often carry sensors, such as viewing-only cameras, that they do not use for automatic control. These sensor model modules can be connected to the physics modules without making an attachment back to the behavior and controls modules.

[1] Tcl/Tk is available at http://www.scriptics.com/

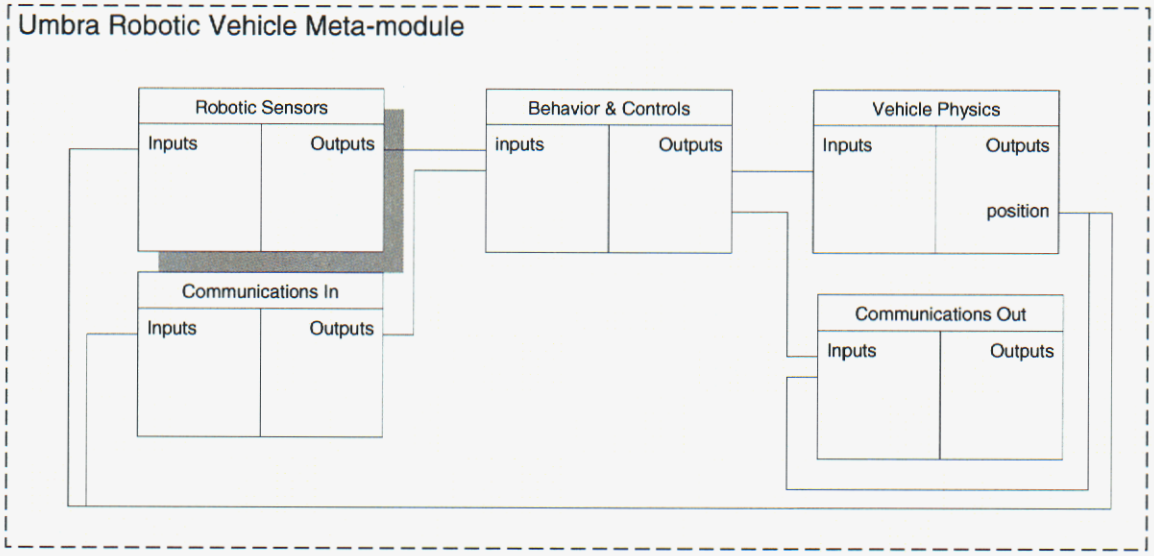


Figure 1: Umbra Robotic Vehicle Meta-module

The Umbra HLA Library

Figure 2 shows a conceptual diagram of how Umbra robot vehicle meta-modules are integrated via HLA for the JVB. Here, Umbra publishes behavior and platform data through separate HLA objects. In addition, for tightly coupled robot systems, Umbra sensors and physics models will subscribe to HLA sensor, environment, and mobility services.

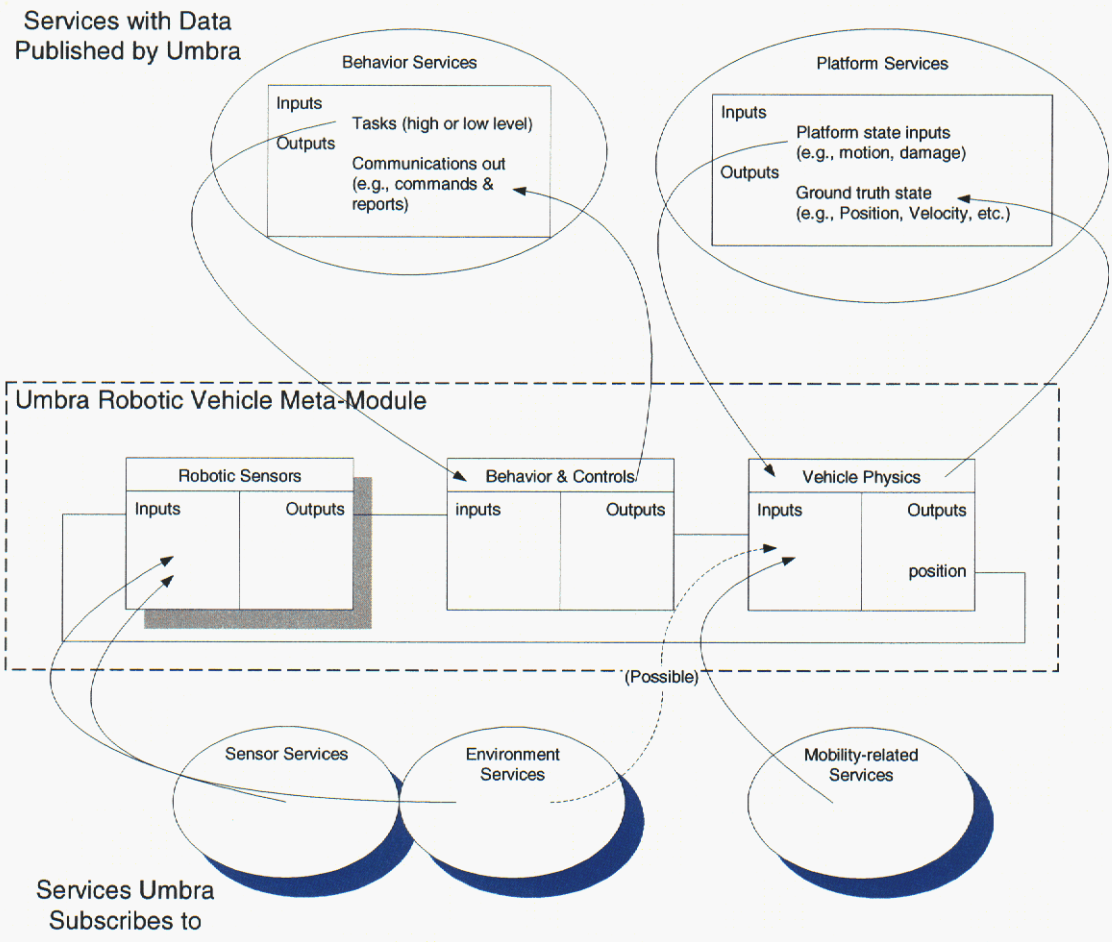


Figure 2: Simplified Diagram Showing Umbra to HLA Integration

Using the HLA Library, models are composed to conform to the federation's interface as defined through its Federation Object Model (FOM). For the JVB, robotic system models are implemented in accordance with the JVB-FOM with separate behavior and platform services. Where it's important, for efficiency reasons, behavior is tightly coupled to sensor input or platform motions within Umbra. At the same time, HLA Objects separately representing Platform and Behavior services are presented to the HLA as separate services. This internal coupling is transparent to the FOM to allow maximum flexibility.

Published services are modeled both internally to Umbra and, for loosely coupled systems, modeled externally through subscribed services. For example, robotic tanks might model mobility and battle damage within Umbra or externally by having Umbra subscribe to services that are outside Umbra.

It is noteworthy that the HLA interface allows the computational portions of the modules to be distributed among separate federates at a component level. For example, behavior services might generate commands, status and high-level reports. Other elements of the HLA simulation environment might transfer task and report data to and from the behavior services while propagating this C4ISR data through the environment. Platforms, sensors, and other objects might be attached, either temporarily or permanently, to one another to represent systems with combined functionality.

Through the HLA interface, other systems are able to instantiate, command, and monitor individual as well as integrated collections of platforms. In addition, Umbra can monitor HLA objects that it does not control. This monitoring is important for allowing Umbra to interact with the entire HLA simulation environment.

Umbra HLA Class Hierarchies

The Umbra HLA implementation utilizes Umbra's *Worlds* abstraction to achieve a close matching of Umbra modules to HLA components. An Umbra module class, called the Ambassador, has been developed for communications management. An Ambassador module is created for each federation that an Umbra application joins. Umbra proxy modules are instantiated to support specific interactions and as one-to-one proxies to specific HLA Object Class instances. While any proxy module can be programmed to send or receive any HLA interaction allowed by the federate, interaction-only proxy modules are typically created to proxy sets of HLA Interaction Classes. Object Class proxy modules are created to proxy individual HLA Object Class Instances.

In addition to being an interface, the Ambassador module is also a factory of all object and interaction proxy modules. The Worlds abstraction is used to provide an added level of control between the Ambassador factory module and the interaction and object proxy modules.

At the implementation level, DMSO's RTI provides an abstract class, called FederateAmbassador, that identifies the callback functions that each federate is obliged to provide. Umbra provides another class, called UmbDynamic, that provides module services including dynamic updating. Umbra HLA library provides an Ambassador class that uses multiple inheritance to combine the functions of FederateAmbassador and UmbDynamic into one class. This inheritance scheme is shown in Figure 3.

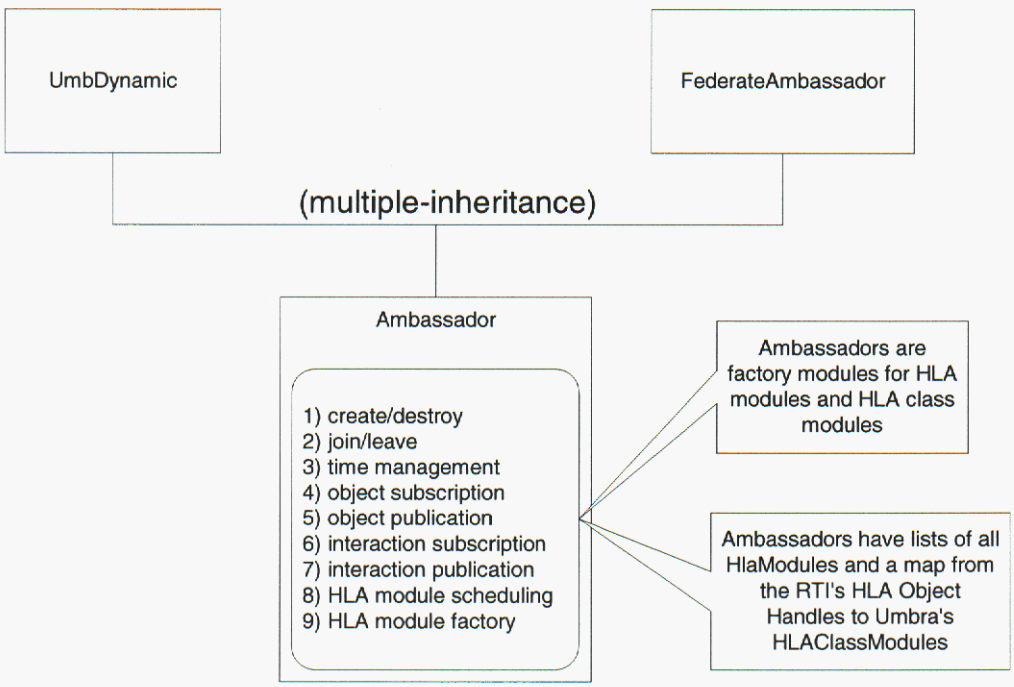


Figure 3: Class Hierarchy that Defines Ambassador

The Umbra Ambassador implements the callback functions required by the RTI software. Specifically, Ambassador implements create/destroy, join/leave and object and interaction subscription and publication. In addition Ambassador provides HLA module factory and scheduling services.

Umbra's HLA library provides a HlaModule virtual class that can send and receive interactions and a HlaClassModule virtual class that, in addition, can provide proxy object references that correspond to specific HLA objects defined in the RTI. HlaModule is derived from UmbDynamic and HlaClassModule is derived from HlaModule. Within an HLA application, derivative classes are built to implement specific functionality required by the federation. Figure 4 shows the class hierarchy of these modules in relation to user modules developed for particular applications.

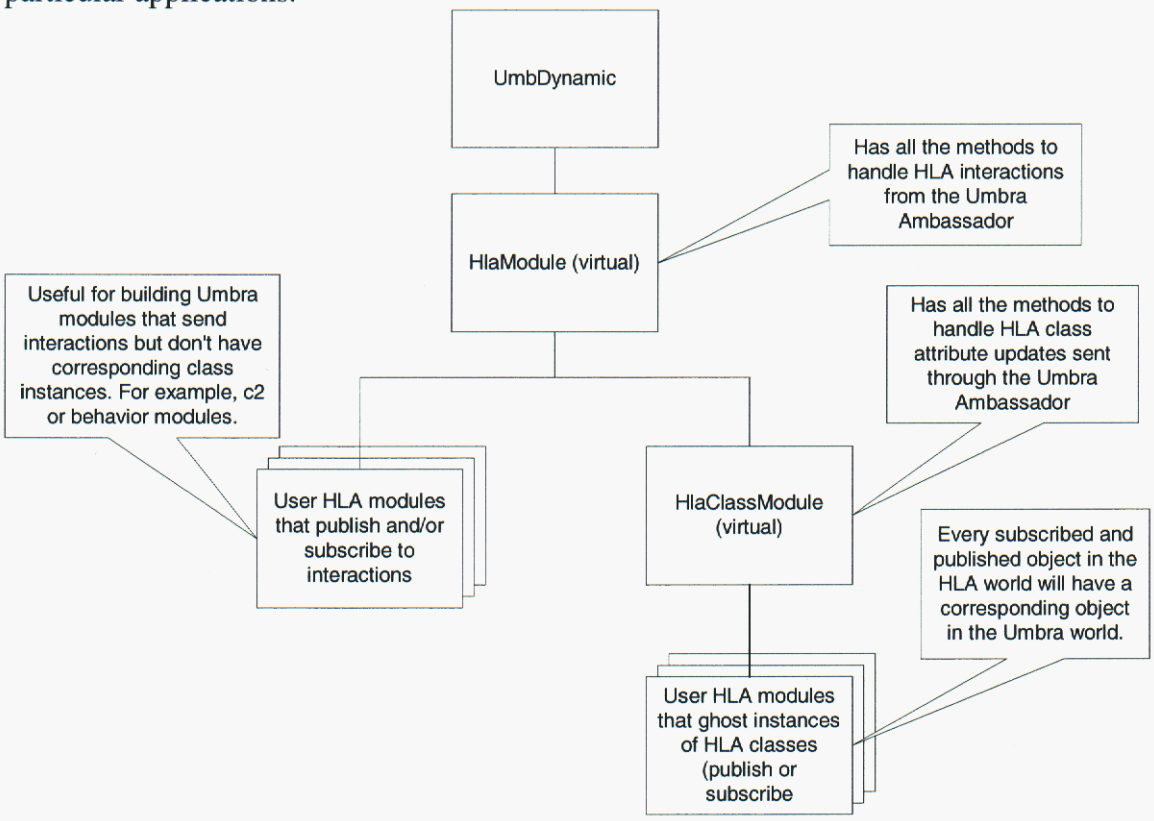


Figure 4: Typical Class Hierarchy for Application-Specific HlaObjects

HlaModule and HlaClassModule implement basic functionality for functions that are required for interacting with an Umbra Ambassador. Derived HLA modules can override this functionality to provide specialized behavior.

Federation Creation & Destruction

Umbra Ambassador modules implement Federation Creation and Destruction and Federate Joining and Leaving as matched pairs of behavior. The Create Federation command can be invoked through a Tcl command that is sent to the Ambassador. Once this command has been called, the Ambassador will assure that a Destroy Federation command will be issued before the Ambassador module is destroyed. Likewise, if the Join Federation command is invoked through its Tcl command, the Leave Federation command will be issued prior to Ambassador destruction. Destroy and Leave can also be invoked through Tcl commands.

Time Management

The HLA provides time management services to coordinate events between simulation environments. HLA time management services have specific design features to allow federate simulators to operate within a well-controlled time disparity band. For example, simulation time among JVB federates is typically held to within a few seconds of each other.

To keep network traffic at a manageable level, typical HLA federates pace time in relatively large steps (in the order of seconds). Conversely, to provide sufficiently high fidelity physics and interactive capabilities, typical Umbra simulations are run with relatively small time steps (in the order of micro seconds). Our Umbra-HLA integration provides a time management and coordination scheme that allows the HLA federates to make large time steps (e.g., greater than 1 second) while the Umbra environment uses small internal time steps (e.g., less than 100 milliseconds). In the limiting case, the design allows Umbra to operate with arbitrarily large time steps designed to keep pace with any federation.

Our solution implements a time base that interpolates federation time to derive a local (interpolated) Umbra time that can be used for physics effects computation and other time-rate sensitive modules. (This time value is available to all Umbra modules within a simulation.) For purposes of discussion, time will be described as being broken into threads that artificially begin whenever Umbra synchronizes its time with the federation's time base.

The Ambassador provides a connector-based interface to its local interpolated time that simulation modules can use for computation. This clock is provided for coarse time synchronization. In a typical simulation, for example, physics modules use the Ambassador's *dt* or delta time connector to determine the integration time over which to perform their computations. Behavior modules use the *time* connector in their event scheduling mechanisms. Modules are free to ignore this clock. For example, simulation control modules typically act on interactions that they receive without regard to the interpolated time base. (They do, however, rely on the RTI to deliver messages within the appropriate time band.)

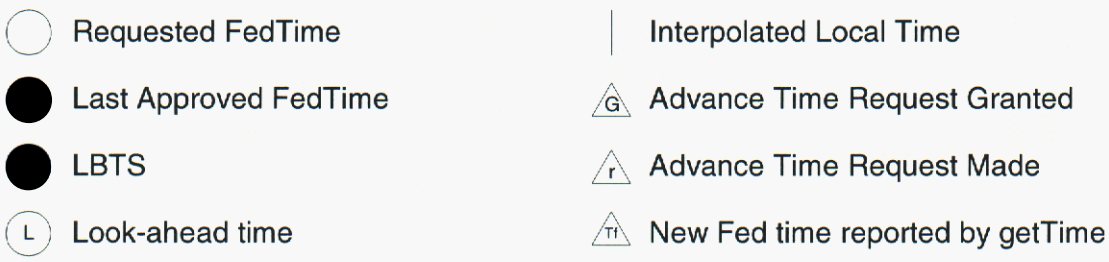


Figure 5: Key for Time Management Drawings

The following sections describe the various time management algorithms that have been built in Umbra. Figure 5 provides a drawing key for the figures used in this description. For clarity, we describe the non-regulating and unconstrained time management case first, then the regulating and constrained case, and finally the non-regulating and constrained cases.

Non-regulating, Unconstrained Umbra Federates

Most of the time, non-regulating, unconstrained federates will not pay attention to the federation's simulation time. For example, real-time simulators don't typically use HLA's time management features. To support real-time applications, the Ambassador reports real-time data on when it is connected to a wall or real-time clock module. In some cases, it may be beneficial to coarsely match Umbra's time base with time being managed by other federates. To allow this potential, the Ambassador provides a time base that roughly tracks the federation's time.

The basic algorithm for non-regulating, unconstrained Umbra federates is to have the Umbra simulation clock try to keep up with the federation's Lower Bounded Time Stamp (LBTS). The algorithms to track time are written with the expectation that the LBTS will be irregular in both the rate or frequency that time changes are reported as well as the time steps that are reported. Figure 6 diagrams this basic algorithm by showing time advancing through separate time threads. It is noteworthy that this algorithm can be overridden by connecting the Ambassador module to an external clock. For example, wall clock modules can be connected to the Ambassador module for real time simulations.

In the typical case, the Ambassador module will generate an interpolated clock tick every 1 to 100 milliseconds and watch the federate for time updates. By monitoring the LBTS through, Umbra will occasionally, (i.e., every 1 to 10 seconds) notice that the LBTS has advanced. If the advance is within reasonable bounds, Umbra will keep running its interpolated clock at its normal pace. If the advance is very far out in the future, Umbra will increase its interpolated time steps to catch up with the federation's LBTS time. (The drawing shows this as Time Warping.) If Umbra's time catches up to the LBTS, Umbra will stop its interpolated clock (set the delta time to zero) while continuing to pulse modules to allow them to process time insensitive events.

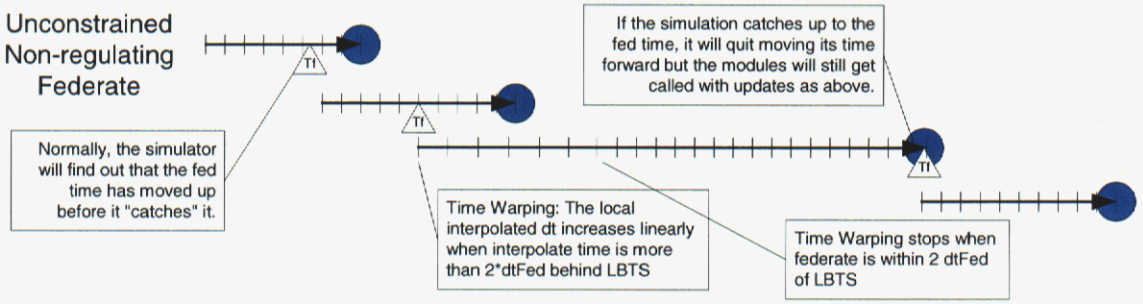


Figure 6: Time Management for Unconstrained and Non-regulating Umbra Federates

The specific algorithm works as follows:

- On initialization, Umbra queries the federation to determine the current LBTS. If the LBTS is infinite, Umbra will assume that no time regulation is being performed and will start ticking its clock from zero time. (Umbra always initializes its Ambassadors with non-regulating and unconstrained time management.)
- At every update, Umbra will check the LBTS to determine how to interpolate local time.
  - If LBTS time is equal to or smaller than Umbra time, Umbra will stop moving its local time forward. (It will continue to provide modules with update functions. Only time-based functions that use the HLA clock will be affected by this *stopping of time*.[2])
  - If LBTS time is within two *dtFed* time steps of Umbra time, Umbra will advance time by *dt*. (*dtFed* and *dt* are Umbra Ambassador class variables.)
  - If LBTS time is greater than two *dtFed* time steps of Umbra time, Umbra will advance time by (1- *(Lbts – umbTime)/dtFed)* * *dt*.

Regulating and Constrained Umbra Federates

Regulating and constrained time management is the next simplest case. Here, Umbra interpolates its local time within a time span that keys off its last federate granted and the next federate requested times. (LBTS is ignored.) Two algorithms have been implemented. The first is time following, where Umbra's local interpolated time follows slightly behind its last time advance request. The second is time leading, where Umbra's local interpolated time follows slightly ahead of its last time request.

Figure 7 diagrams Umbra's time following algorithm for regulating and constrained time management. To understand the algorithm, first assume Umbra has been running and has recently passed a time advance grant. After a short time, Umbra will notice that its Umbra time has overrun its lookahead time. At that point, Umbra will make a time advance request (first triangle) and keeps stepping its local time forward at its regular pace. (This puts Umbra's time slightly ahead

[2] An algorithm to slow Umbra time as it approaches LBTS time may be implemented at a later date.

of the last time request grant.) Some time later, the RTI will grant the time advance request, placing Umbra onto a new time thread. (This also puts Umbra's time slightly behind the last time request grant.) At that point, Umbra will continue to interpolate its local time and repeat the basic algorithm. A special case arises, when Umbra's local time catches up to its last requested plus lookahead time. When this occurs, Umbra holds its simulation clock in the same way as it does with non-regulating unconstrained time management.

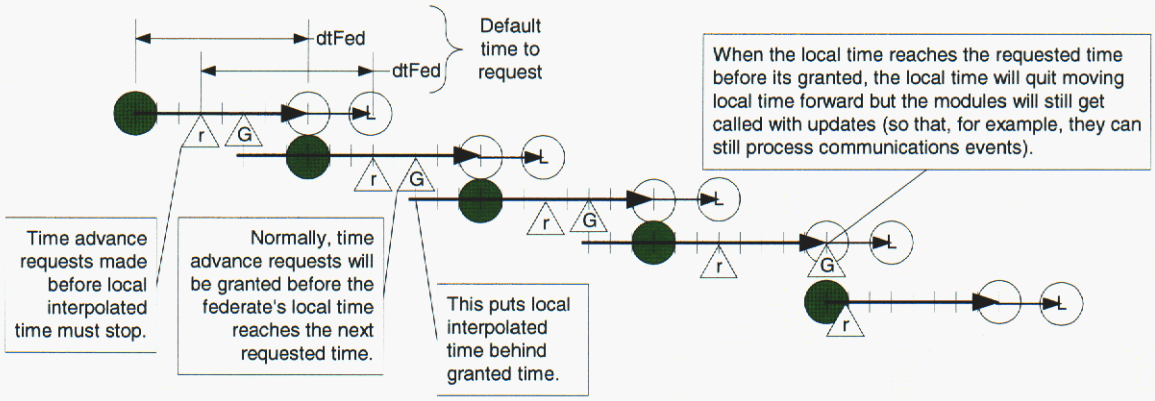


Figure 7: Time Management for Constrained and Regulating Umbra Federates

The specific algorithm works as follows:

- As always, the Ambassador checks the federation's current LBTS on initialization. If the LBTS is infinite, Umbra assumes that no time regulation is being performed and will start ticking its clock from zero time.
- When the Ambassador first requests regulating and constrained time management, it makes a time advance request one *dtFed* larger than its local interpolated time.
- At every update, Umbra monitors whether it has been granted its time request or is approaching time to request another time advance.
  - When Umbra's time is more than one lookahead increment past the last time grant, Umbra will make a time advance request.
  - If Umbra's time is not greater than the last time request or the lookahead time,[3] it will increment its local time by *dt*.

Umbra's time leading algorithm for regulating and constrained time management is similar to the time following algorithm. To understand the algorithm, again assume Umbra has been running and has recently been granted a time advance. Also assume that its local interpolated time is close to the time grant. From here, Umbra will continue to interpolate its local time forward. When its local time becomes greater than one of its federation time steps plus its lookahead past the last time grant, Umbra will make a time advance request with the time it just

[3] Currently, Umbra is programmed to stop advancing its time when it reaches the last time request time.

passed and continue stepping its local time forward at its regular pace. (This puts Umbra's time slightly ahead of the last time request.) Some time later, the RTI will grant the time advance request, restarting the scheme and Umbra will continue to interpolate its local time and repeat the basic algorithm. A special case arises when Umbra's local time reaches two of its federation time steps greater than the last granted time. (This is also one of its federation time steps beyond its last time request.) When this occurs, Umbra stops its simulation clock in the same way as it does with non-regulating unconstrained time management and waits for a time advance grant.

Non-regulating, Constrained Umbra Federates

As with non-regulating unconstrained time management, Umbra's non-regulating constrained time management algorithm has Umbra pacing itself to keep in step with the federation's time, as measured by the current LBTS. Here, however, Umbra makes time advance requests to assure that time stamped events (sent from regulating federates) are presented to Umbra in a time base that matches its own sense of time. Figure 8 diagrams this algorithm.

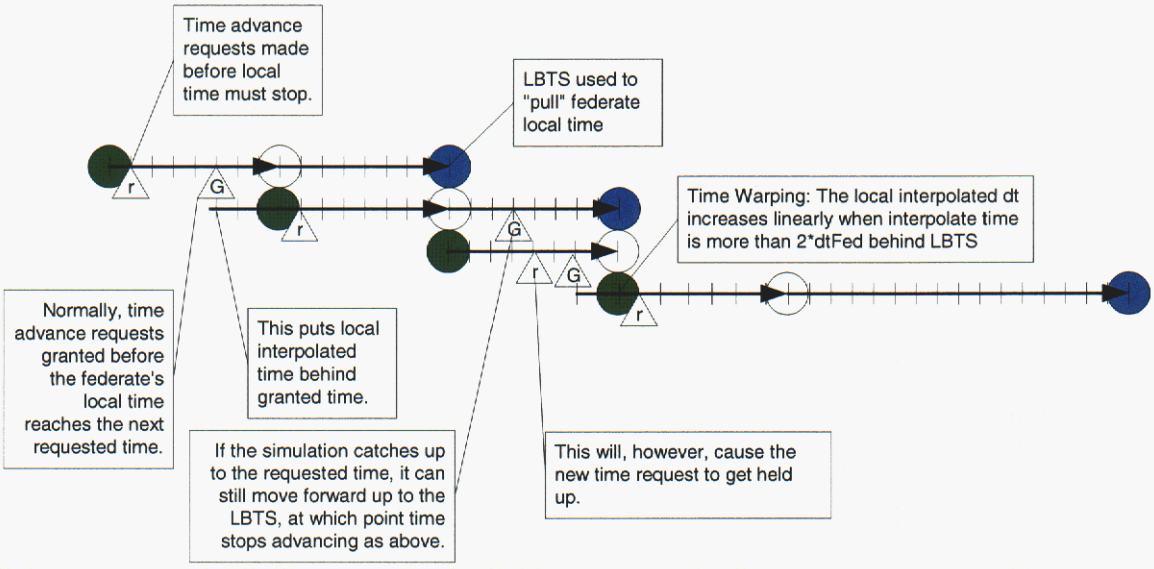


Figure 8: Time Management for Constrained and Non-regulating Umbra Federates

The specific algorithm works as follows:

- As always, Umbra will have initialized its Ambassadors for non-regulating and unconstrained time management and be roughly in synch with the federation's LBTS time.
- When Umbra first requests constrained time management, it makes a time advance request one *dtFed* larger than its local interpolated time.
- At every update, Umbra will query the LBTS to determine how to interpolate local time.

    - If LBTS time is equal to or smaller than Umbra time, Umbra will stop moving its local time forward. (It will continue to provide modules with update functions. Only time-based functions that use the HLA clock will be affected by this *stopping of time*.)
    - If LBTS time is within two *dtFed* time steps of Umbra time, Umbra will advance time by *dt*. (*dtFed* and *dt* are UmbAmbassador class variables.)
    - If LBTS time is greater than two *dtFed* time steps of Umbra time, Umbra will advance time by (1- (*Lbts – umbTime)/dtFed) * dt.*
- In addition, at every update Umbra monitors whether it has been granted its time request or is approaching time to request another time advance. When Umbra's time is more than one lookahead increment past the last time grant, Umbra will make a time advance request.

Object Management

Relationship Between Umbra and HLA Objects

Umbra simulations publish and subscribe to RTI interaction parameter and object attribute data through proxy modules. Each HLA Interaction Class or Object Class that an Umbra simulation publishes or subscribes to is represented within Umbra with an application-specific interface proxy module. These modules proxy the HLA by maintaining local state data concerning the attributes and parameters and by providing mechanisms for moving HLA state and interaction data between other Umbra modules and the HLA environment. Interface proxy modules do not typically provide modeling services. Rather, they communicate with other modules within the Umbra environment that in turn provide modeling services.

For example, a robotic system might be modeled within Umbra using a variety of connected modules (e.g., see Figure 1). Various aspects of this model may have HLA counterpart objects. Key modules that make up the Umbra robot are connected to Umbra HLA proxy modules to transfer state data between the Umbra robot modules and the HLA world. It should be noted that this separation is not a requirement, but rather it is done for convenience and rapid programming.

HlaModule is the base or virtual proxy class for handling interactions. HlaClassModule, which is derived from HlaModule, adds the ability to proxy instances of HLA Object Classes. Application-specific modules are built by inheriting from these modules. All HlaModule modules can be programmed to send or receive any interaction (providing, of course, that federate has subscribed to the interaction or the RTI has granted the federate permission to publish). In fact, more than one module can be programmed to receive the same interaction. Here, the Ambassador module provides a copy of the interaction data to each module so that it can perform its separate processing function.

The HlaClassModule is used to proxy instances of HLA Object Classes. Separate application-specific HlaClassModules are instantiated to proxy individual object instances created in the HLA federation. Proxy modules are created both for objects that the federation publishes and for objects that the federation discovers through subscription. For example, a typical Umbra application that publishes 50 Platform objects and discovers another 200 Platform objects in a JVB federation will create 250 Platform proxy modules (derived from HlaClassModule) to proxy these objects.

Ambassador Factory Services

Umbra provides an interface for allowing Umbra modules to create other Umbra modules. This capability is called a Factory Method[4] and is an underlying mechanism for creating Umbra worlds. The Ambassador uses this facility to create all HLA proxy modules.

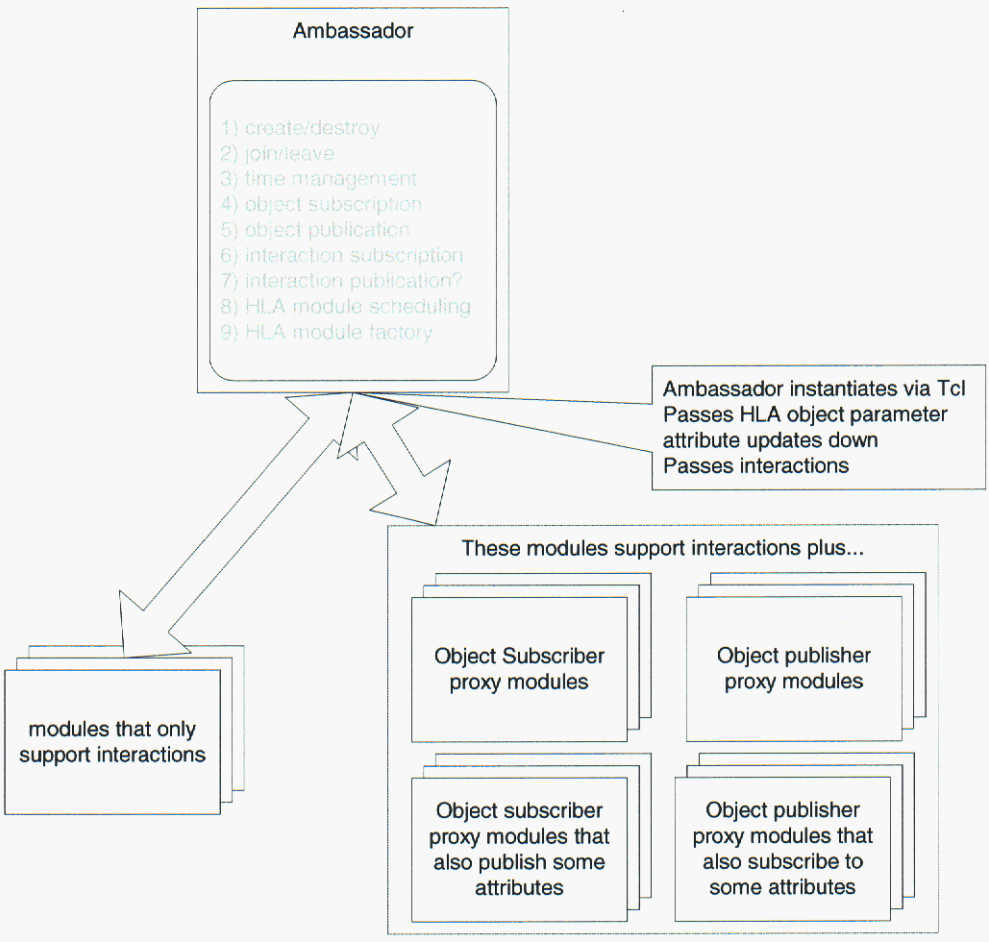


Figure 9: Ambassador Factory Services

Figure 9 shows five basic types of HLA proxy modules objects that Umbra's Ambassador can create. These are:

- Proxy modules that only support interactions.
- Proxy modules that are created to publish instances of HLA Object Classes.
- Proxy modules that are created to subscribe to attribute data from externally published instances of HLA Object Classes.

[4] Gamma, Helm, Johnson, and Vlissides, **Design Patterns, Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995, pp 107-116.

- Proxy modules that are created to subscribe to attribute data from externally published instances of HLA Object Classes and then acquire control of and publish attribute value data.
- Proxy modules that are created to publish instances of HLA Object Classes, give up control of instance variables and subscribe to attribute value data.

In addition to creating objects, the Ambassador keeps track of and manages the objects that it creates. For example, the Ambassador passes HLA data to its proxy modules, allocates specific time intervals during which HlaObjects can publish data, and destroys Subscription objects when the corresponding HLA Objects are destroyed.

Proxy Modules that Support HLA Interaction Classes

Figure 10 shows the software interactions involved in creating basic HLA proxy modules. As was noted earlier, basic modules can only support interactions. It is typical to utilize basic proxy modules as unique interfaces to specific groups of interactions. For example, in JVB, one proxy module was created to handle the remote_create interaction while another was created to handle propagated commands or orders from the Command, Control and Communications (C3) grid.

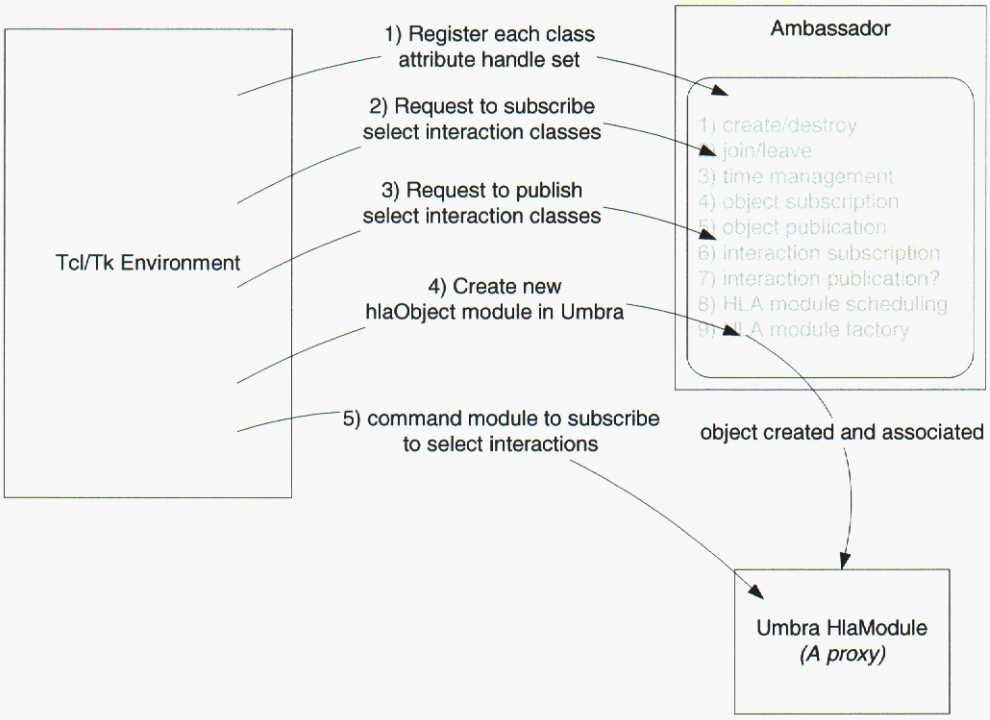


Figure 10: Creating basic HLA proxy modules

The process for publishing interactions proxy modules follows the following steps:

1. The application registers each class-attribute handle set (`cName` and `attributeNames`) through Umbra with the RTI and stores the class and

attribute handles within the Ambassador module. Assuming that the Ambassador module is named amb, the registration command for interactions is of the form:

```
amb getRtiInteractionHandles className attributeNames
```

2. The application may request to subscribe to specific Interaction Classes in the HLA world. The request to subscribe command is of the form:

```
amb subscribeInteractionClass $interactionName
```

or, if the application wishes to restrict interactions to a specific space,

```
amb subscribeInteractionClassWithRegion $interactionName \
    $routeSpace
```

3. The application may also request permission to publish specific Interaction Classes in the HLA world. The request for permission to publish is as follows:

```
amb publishInteractionClass $interactionName
```

4. The application has the Ambassador create an application-specific Umbra proxy module. In the JVB, a typical module was called Order and the proxy module creation command was of the form:

```
amb hlaProxyModule oName
```

5. The proxy module is then typically commanded to individually subscribe to specific interactions. (The module, in turn, commands the Ambassador to associate the receipt of these interactions to that module.) It is also typical to provide the name of a Tcl script that will be called when the interaction is received. In the JVB, associations for the Order module were done with the commands:

```
$name subscribeInteraction \
      Networking.Communication.Command.Move
$name subscribeInteraction \
      Networking.Communication.Command.FollowVehicle
$name subscribeInteraction \
      SimulationService.FireEngagement.DamageReport
```

Publication-based HLA Object Class Proxy Modules

Figure 11 shows the software interactions involved in creating HLA class proxy modules for HLA Object instances that Umbra publishes, including those that give up control of instance variables to other simulators and subscribe to their values. Generally, an Umbra application requests permission to publish an HLA class and then creates or publishes a number of class instances by creating Umbra HlaClassModules as proxies to their corresponding HLA class instances.

The request to publish is typically done at the beginning of the program and object instantiation is done as needed (e.g., as platforms are created). As a factory, the Ambassador is responsible for creating the proxy modules. Ambassadors directly instantiate objects upon receipt of an object creation command (generally invoked through Tcl). Once the proxy module is created, the Ambassador returns a pointer to the new proxy module so that it can be further referred to from within the Tcl environment. Typically, these HlaClassModules are connected to other Umbra modules to form a complete device model.

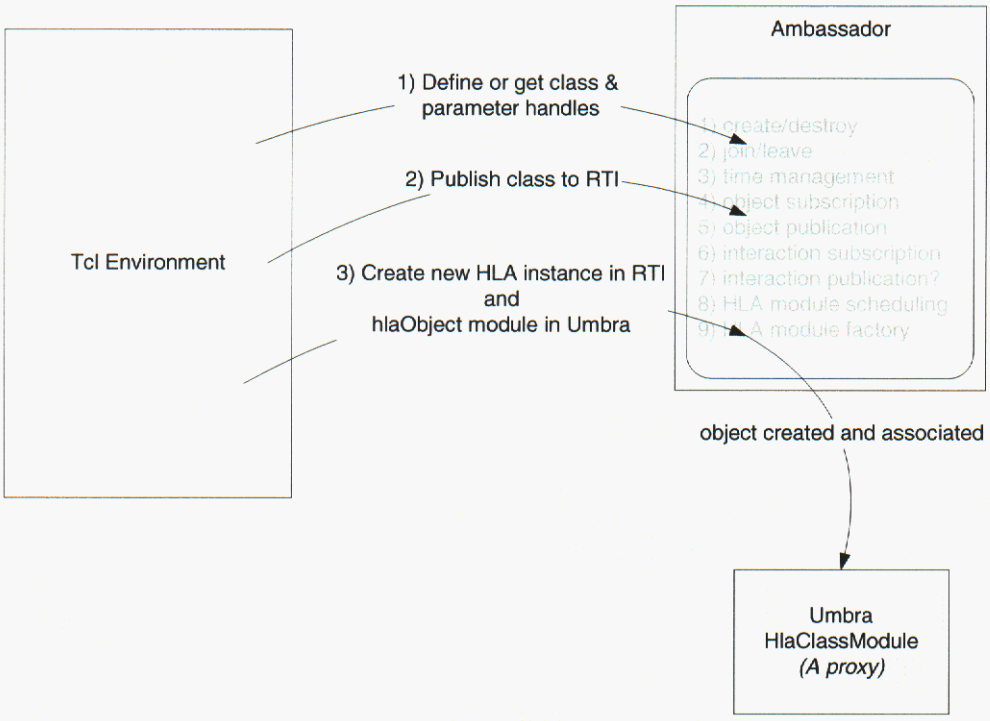


Figure 11: Creating HLA publisher and publisher/subscriber proxy modules

The process for publishing Object proxy modules follows the following steps:

1. The application registers each class-attribute handle set (`cName` and `attributeNames`) through Umbra with the RTI and stores the class and attribute handles within the Ambassador module. Assuming that the Ambassador module is named amb, the registration command for interactions is of the form:
   ```
   amb getRtiObjectHandles className attributeNames
   ```
2. The application requests to publish an object class to the HLA world. The request to publish command is of the form:
   ```
   amb publishObjectClass cName
   ```
3. The application has the Ambassador create an Umbra proxy module. During creation, Umbra also publishes the object instance to the HLA federation and establishes the needed mappings to support all proxy functions. For HLA Object Classes, the HLA object name and Umbra modules can be named differently. The instance creation command is of the form:
   ```
   amb ProxyModuleType cName modType oName
   ```

Subscription-based HLA Object Class Proxy Modules

Figure 12 shows the software interactions involved in creating proxy modules for instances of HLA Object Classes that Umbra subscribes to. Generally, Umbra registers (sets up the interaction handles) then subscribes to an HLA class. Later, the RTI calls Umbra with an application-provided callback function which, in turn, calls a Tcl procedure which creates a connected set of meta-modules for monitoring particular HLA class instances and other representing the data in the simulation.

Registration, subscription and instantiation are done at different times (i.e., with different calls to Umbra's Ambassador). To provide the highest-possible degree of flexibility, Umbra's subscription-based proxy modules are not instantiated or destroyed until after the corresponding HLA Object Class instances come into existence (i.e., are published or destroyed by another federate). Rather, the Ambassador subscribes to an HLA Object Class and then, when another federate publishes an HLA Object Class instance, the RTI informs the Ambassador that the HLA Object of the subscribed Class has been created or destroyed in the HLA world. Once the Ambassador is informed that an HLA Object Class instance has come into existence, it performs the necessary calls to create the corresponding proxy module as well as any internal models that interact with or represent the instance.

Umbra HlaObjects are typically connected to other Umbra modules. The code to connect modules is typically defined within Tcl scripts and must be executed after the subscription instances are created. To provide this capability, Ambassador is given a callback script that, when executed, uses similar logic as Publication Objects to ask the Ambassador to create the subscription object. This script then uses the handle returned by the Ambassador from the discovery process to build and connect the other modules that the subscribed object interacts with.

The interaction generally follows the following steps:

1. The application registers each class-attribute handle set (`cName` and `attributeNames`) through Umbra with the RTI and stores the class and attribute handles within the Ambassador module. Assuming that the Ambassador module is named amb, the registration command is of the form:
   ```
   amb getRtiInteractionHandles className attributeNames
   ```
2. The application requests to subscribe to the class and at the same time provides a callback script (`cbScript`) and subscribes to the class, providing a callback script (`cbScript`) and list of argument values that should be returned in the callback (`args`). Umbra then subscribes to the class through the RTI. The subscription command is of the form:
   ```
   amb Subscribe cName cbScript args
   ```
3. Once the RTI informs Umbra that an HLA Object of the subscribed Class has been published by another federate, Umbra schedules the Tcl interpreter to execute the callback script (`cbScript`) while providing the name of the subscribed object as published in the HLA world (`oName`) and the unmodified list of arguments. The callback command is called within Umbra as follows:
   ```
   after delayTime "cbScript oName args"
   ```
4. After Umbra completes its scheduling operations, the Tcl interpreter calls the callback script.
5. Typically, the callback script has the Ambassador create an hlaObject designed to utilize the subscribed data and other modules that model the subscribed object. Callback scripts are typically of the form:
   ```
   proc cbScript {oName {args ""}} {
   ```

```
    amb modType $oName
    … # create other modules oMod
    $oName -connect dataOut $oMod dataIn
    …
}
```

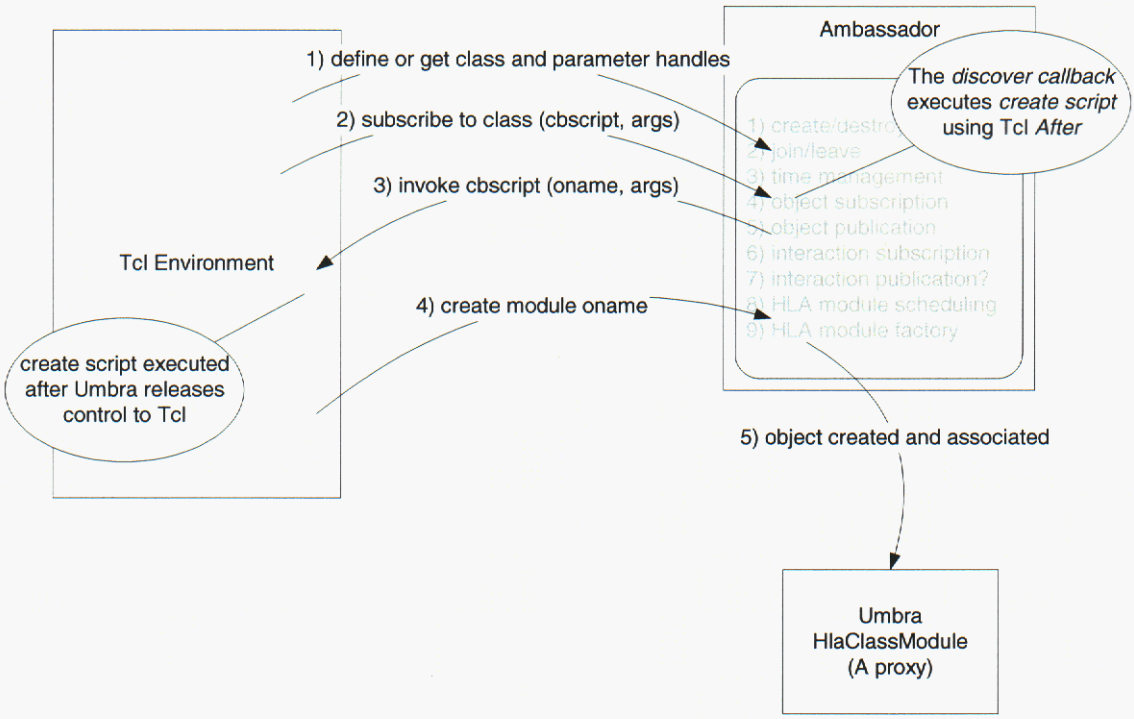


Figure 12: Creating HLA subscriber proxy modules

HLA Data Exchanges

The Ambassador coordinates all data exchanges between Umbra and its associated HLA Federation. It uses the following update logic to perform this data exchange:

1. At a user-adjustable frequency, the Ambassador walks HLA proxy module list and sends each an *Ambassador Update* message.
   - The Ambassador update message frequency is equal or lower to Umbra's main Update frequency.
   - Within the call, each Umbra HLA object can make any number of publish-type calls to the RTI.
2. After the last proxy module has been updated, the Ambassador does an RTI tick. The tick allows the RTI to make its callbacks. Thus, during this and any other tick, the RTI may make several callbacks to Ambassador. The most typical is a call to process an interaction or update attributes.
   - Within these RTI-invoked callback functions, the Ambassador figures out which Umbra proxy modules get the message and then invokes a standard callback message to that specific Umbra HLA proxy module.
   - Typically, the proxy module then caches the data and quickly returns. (Because the RTI is not reentrant modules cannot call the RTI during the RTI callback. A safe programming strategy and encouraged

practice in Umbra is to minimize the processing within the callback. This is practice is also encouraged in the RTI programming guides.)

- Some modules cache the data in their instance variables. Later, when the Ambassador sends an *Ambassador Update* message, these modules finish processing their data.
- Other modules form a Tcl callback function and post the function, possibly with some delay, in the Tcl event loop. Later, the Tcl event loop processes the function (in a way that is guaranteed to be outside the RTI callback).

Figure 13 shows the basic flow of information during a typical Ambassador Update cycle. On its *Umbra Update*, the Ambassador can send each of its HlaObjects an *Ambassador Update*. The HlaObjects then, in turn, post messages to the RTI. Finally, after the Ambassador finishes updating its HlaObjects, it issues a tick to the RTI, upon which the RTI starts issuing callbacks. (See discussion on Figure 14 below.)

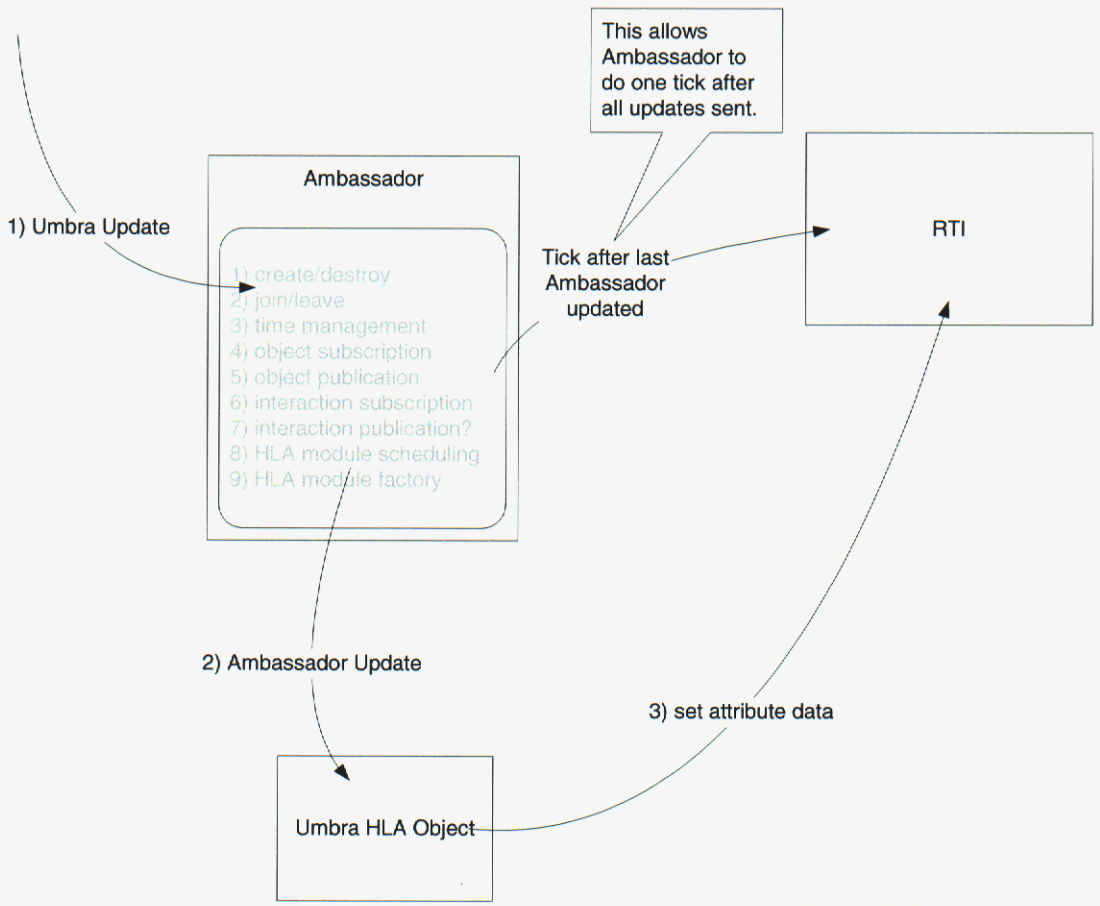


Figure 13: How the Ambassador Coordinates HLA Data Publication

Figure 14 shows how the Ambassador processes callbacks from the RTI. Callbacks are processed while the RTI executes its tick function. As just described, the Ambassador issues a tick after updating its last HlaObject. During

the tick, the RTI may repeatedly call the Ambassador's callback functions. Within the callback functions, the Ambassador relates the object handle provided by the RTI with the appropriate Umbra HlaObject, calls its standard callback method and passes appropriate data. The Umbra HlaObject then, in turn, performs the minimum computation needed to cache the data. Later, when the HlaObject receives its *Umbra Update*, it processes the data for presentation to the other modules.

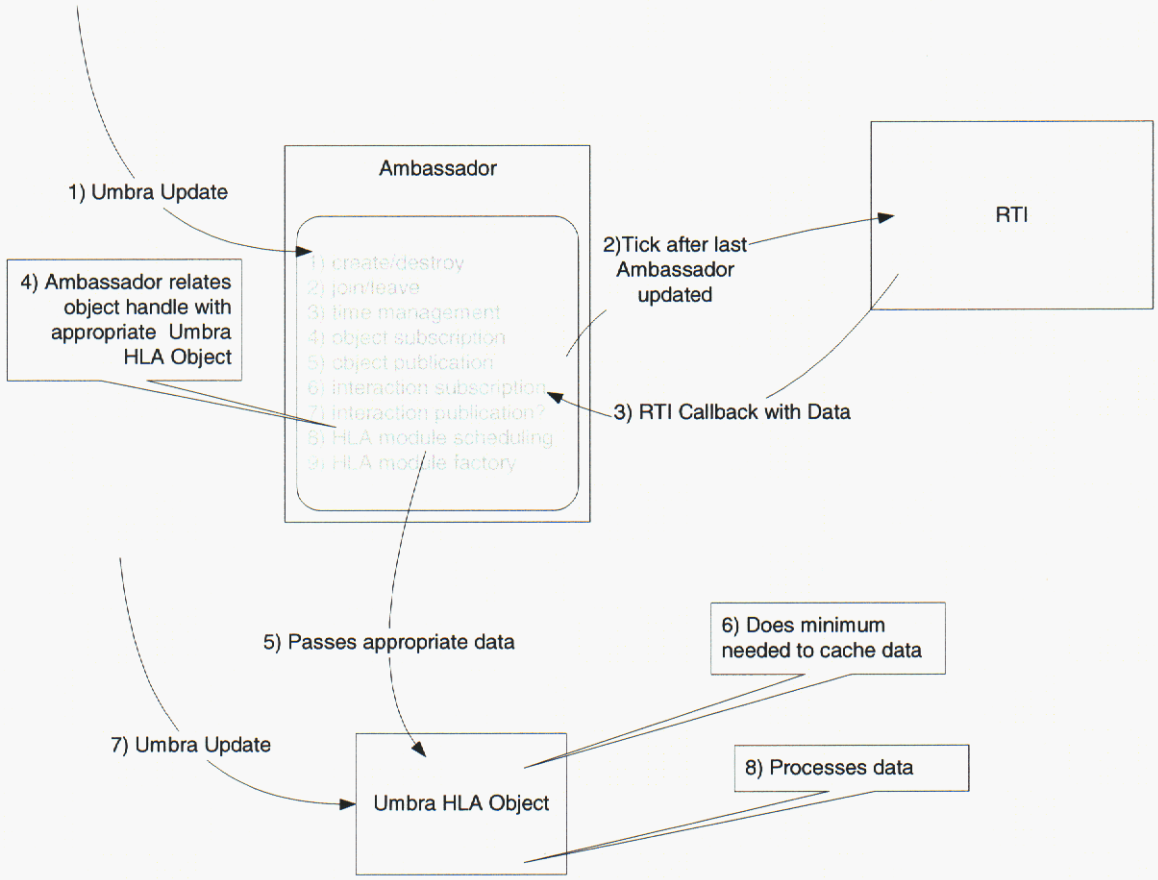


Figure 14: How the Ambassador Coordinates the Receipt of HLA Subscription Data

Interaction Management

Interactions are one-time events that are sent through the RTI. As with object value updates, interactions are initiated or sent from proxy modules through the Ambassador to the RTI. Similarly, the Ambassador receives interactions from the RTI and dispatches them to proxy modules that have previously registered interest or subscribed to the interactions.

Publishing is the three-step process shown in Figure 15. Prior to publishing an interaction, the Ambassador must obtain interaction handle data from the RTI and inform the RTI that it intends to publish each particular type of interaction. These functions are initiated through the Tcl environment as calls to the Ambassador

module. Later, any proxy module can obtain interaction class and variable handles and send the interaction via the Ambassador through the RTI. The Ambassador's getInteraction, getParameterNames, and sendInteraction methods are used here.

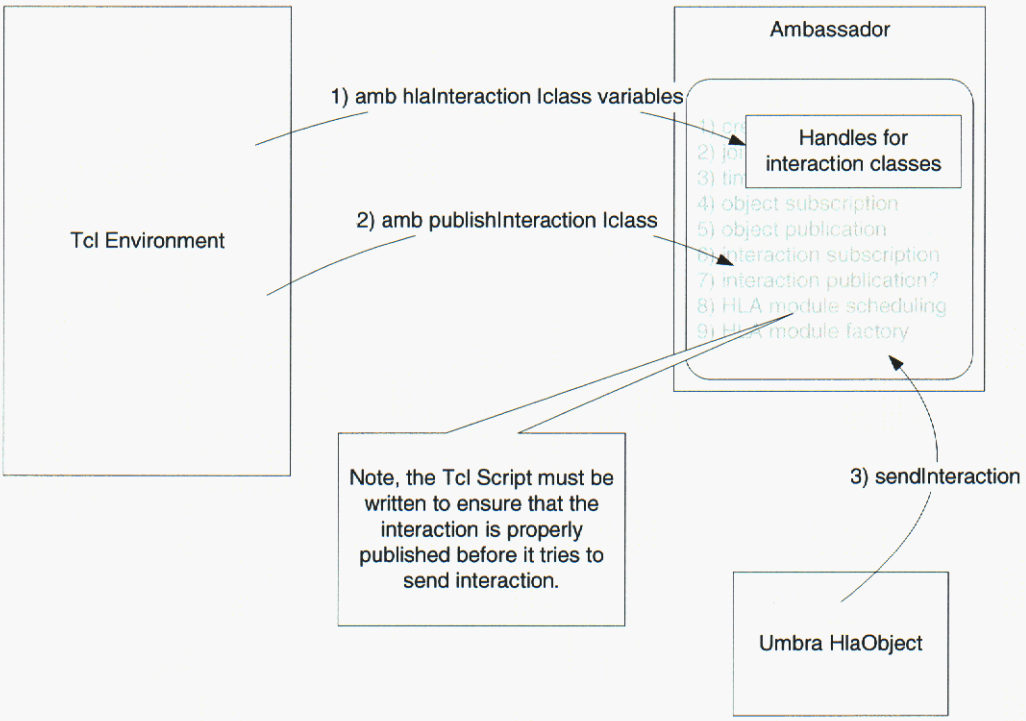


Figure 15: Publishing Interactions

Subscribing is also a three-step process and is shown in Figure 16. Prior to subscribing to an interaction, the Ambassador must obtain interaction handle data from the RTI and inform the RTI that it wishes to subscribe to each particular type of interaction. (The first step is only performed once when the Ambassador both subscribes and publishes.) These functions are initiated through the Tcl environment as calls to the Ambassador module.

Later, individual proxy modules subscribe to the Umbra Ambassador for *copies* of these interactions. These subscriptions can be generically subscribed through the default Tcl subscribeInteraction method (as shown in Figure 16) or through internal code within specialized proxy modules. Tag data values can be used to further constrain which interactions the Ambassador will later pass or dispatch to each proxy module.

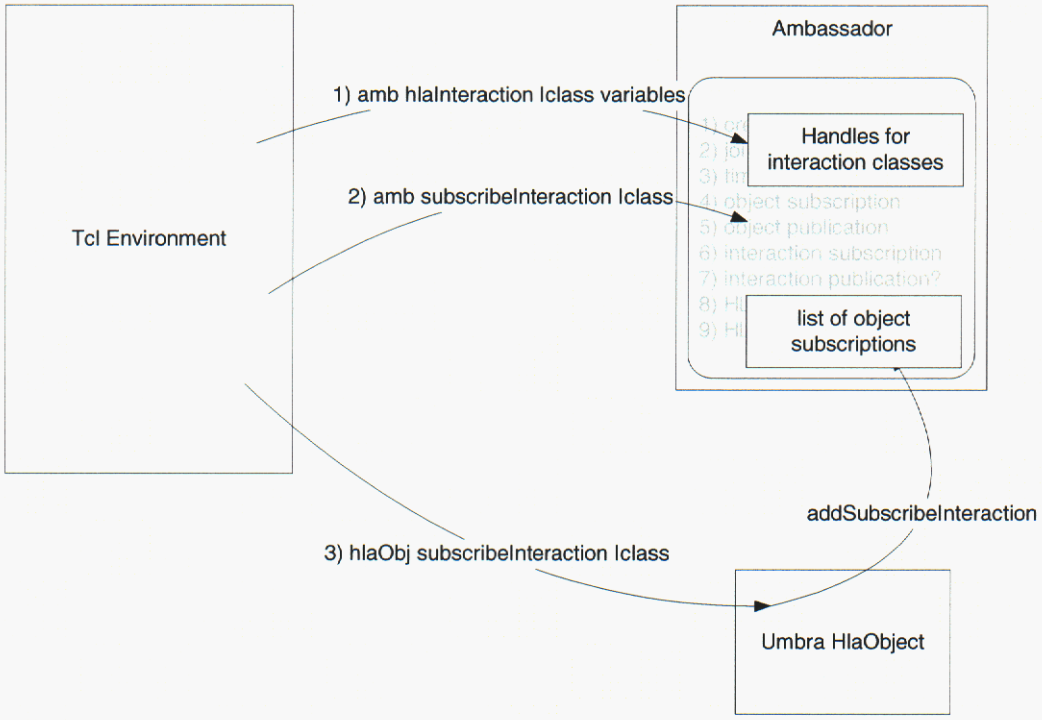


Figure 16: Subscribing to Interactions

Figure 17 shows how interactions are received and dispatched to proxy modules. The RTI automatically calls the Ambassador's receiveInteraction callback when any subscribed interaction is sent through it. (Special features within the RTI can further constrain subclasses of interactions.) Within this callback, the Ambassador dispatches the interaction to each proxy module that has subscribed to the interaction class. (This function also uses tag data to restrict which objects process the data.)

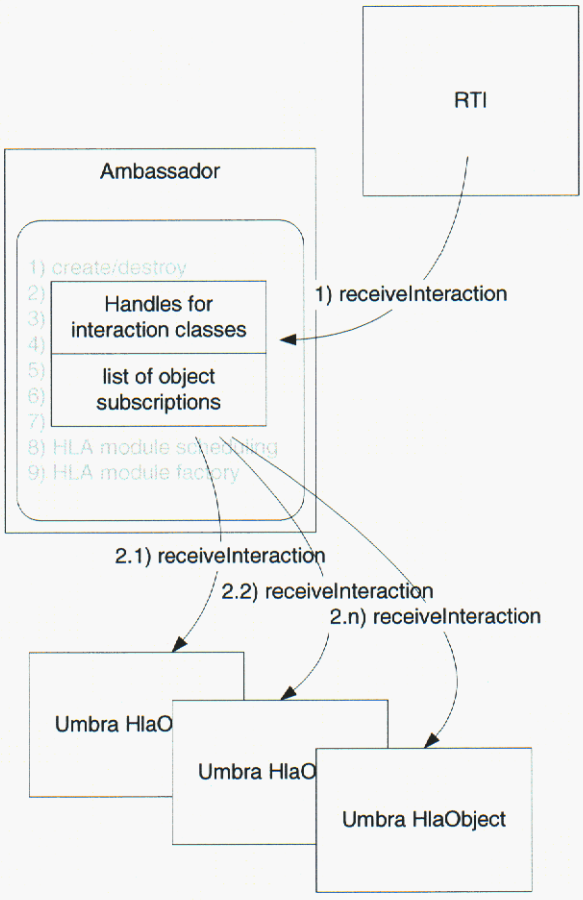


Figure 17: Receiving Interactions

Suggested Reading

The following reports and books are recommended.

1. Buschmann, F; Meunier, R; Rohnert, H; Sommerland, P, **Pattern Oriented Software Architecture: A System of Patterns**, John Wiley & Son Ltd, 1996.

2. Gottlieb, EJ; Harrigan, RW; McDonald, MJ; Oppel, FJ; Xavier, PG **The Umbra Simulation Framework**, June 2001, Sandia Internal Report, SAND2001-1533.

3. Small, DE; Gottlieb, EJ; Edlund, K; Slutter, C; **A Design Patterns Analysis of the Umbra Simulation Framework**, October 2000, Sandia Internal Report, SAND2000-2380.

4. Kuhl, F; Weatherly, R; Dahmann, J; **Creating Computer Simulation Systems; An Introduction to the High Level Architecture**; Prentice Hall, 1999, ISBN 0-13-022511-8.

5. **High Level Architecture Run Time Infrastructure Programmers Guide**, Department of Defense, Defense Modeling and Simulation Office. (http://www.dmso.mil).

DISTRIBUTION:

1	MS1002	J. Langheim, 15200
1	MS1004	R. Harrigan, 15221
1	MS1004	P. Bennett, 15221
1	MS1004	S. Gladwell, 15221
2	MS1004	M. McDonald, 15221
1	MS1004	E. Gottlieb, 15221
1	MS1004	F. Oppel, 15221
1	MS1004	R. Peters, 15221
1	MS1004	B. Rigdon, 15221
1	MS1004	D. Schoenwald, 15221
1	MS1004	J. Trinkle, 15221
1	MS1004	P. Xavier, 15221
10	MS1004	ISRC Library, 15221
1	MS1010	M. Olson, 15222
1	MS1010	D. Small, 15222
1	MS1125	K. Miller, 15252
1	MS1006	L. Shipers, 15272
1	MS1170	R. Skocypec, 15310
1	MS1188	C. Lippitt, 15311
1	MS1188	E. Parker, 15311
1	MS1188	J. Wagner, 15311
1	MS0830	C. Forsythe, 15311
1	MS1188	S. Tucker, 15311
1	MS1188	R. G. Abbott, 15311
1	MS1176	D. Anderson, 15312
1	MS1176	R. Cranwell, 15312
1	MS1176	T. Calton, 15312
1	MS1176	D. Miller, 15312
1	MS0986	M. Platzbecker, 2664
1	MS0780	S. Jordan, 05838
1	MS1137	O. Bray, 06534
1	MS0318	M. Boslough, 9212
1	MS0188	C. Meyers, 01030
1	MS9018	Central Technical Files, 8945-1
2	MS0899	Technical Library, 9616
1	MS0612	Review & Approval Desk, 9612 For DOE/OSTI